CS 271 - Project 0110

James Le

April 11, 2017

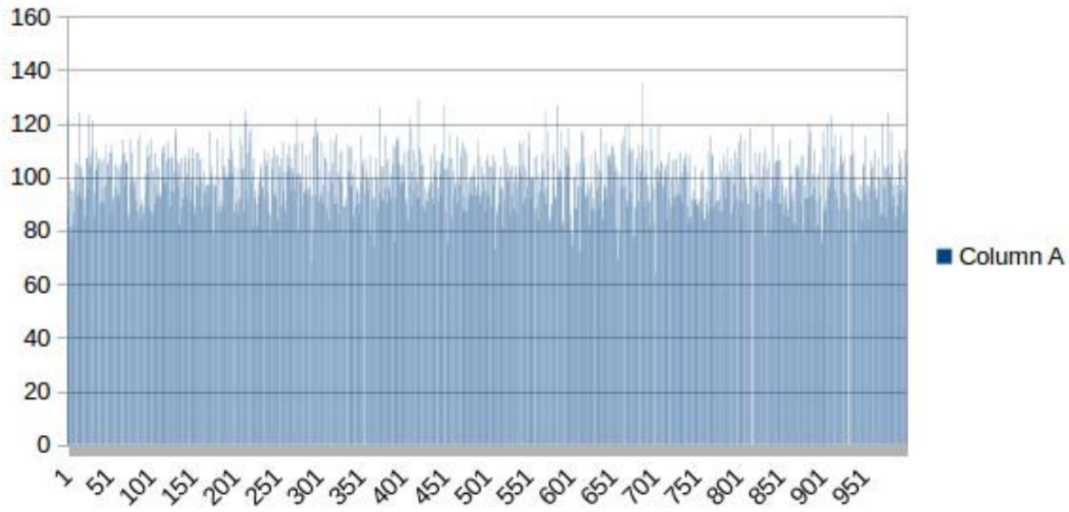
In the hash function, I first check the length of the string in order to customize how the hash function will calculate the slot for a particular string:

1. In the case that the string is of length 1, the ASCII value of that particular character is modified by the number of slots and the resulting value is the slot number the object is assigned to.

2. In the second case, the string has a length of 2, and in this case both of the letters in the string are translated into their respective ASCII codes and added together. The resulting value is multiplied by the ASCII code of the first character in the string and then modified by the number of slots.

3. In the final case, the string has a length that is greater than 2. I begin by creating separate seeds, of which are large prime numbers. Next, I first iterate through the characters in the string, and update the value of a variable by itself multiplied by the ASCII value of the character being iterated on, multiplied by the first seed, and finally modified by the second seed. This is then done again for the same string, for all characters in the string except the final one. These 2 values are then multiplied together, and divided by the string size cubed, and then modified by the number of slots to return the slot number for a string.

   After placing every word in the built-in dictionary into a HashTable object, I then wrote out the size of each entry in the table to a file. Once this was complete, I then created a spreadsheet with these values and calculated the standard deviation of them. I found the standard deviation to be 9.9255, and the graph of the size of each slot is shown here:



```
// hash.h
// James Le
// Project 0110
// CS 271 - Data Structures

#ifndef HASHTABLE
#define HASHTABLE

#include <iostream>
#include "list.h"

template <class KeyType>
class HashTable
{
public:
  HashTable(int numSlots);
  HashTable(const HashTable<KeyType>& h);
  ~HashTable();

  KeyType* get(const KeyType& k) const;
  void insert(KeyType *k);
  void remove(const KeyType& k);

  std::string toString(int slot) const;

private:
  int slots;
  List<KeyType> *table;
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const HashTable<KeyType>& ht);

class Empty{ };
class Key { };
class Index { };

#endif

#include "hash.cpp"
```

```
// hash.cpp
// James Le
// Project 0110
// CS 271 - Data Structures

#include <iostream>
#include <stdlib.h>
#include <string>
#include <sstream>
#include "test.cpp"

using namespace std;

/*=====================================================================
HashTable(int numSlots)       // Default Constructor
Preconditon: None
Postcondition: Creates an empty hashtable instance of size numSlots
=====================================================================*/
template <class KeyType>
HashTable<KeyType>::HashTable(int numSlots)
{
    slots = numSlots;
    table = new List<KeyType*>[numSlots];
}

/*=====================================================================
HashTable(const HashTable<KeyType>& h)    // Copy Constructor
Precondition: Must be a given hashtable
Postcondition: Traverses the hash table and makes a copy of its values
  to transfer to another hash table
=====================================================================*/
template <class KeyType>
HashTable<KeyType>::HashTable(const HashTable<KeyType>& h)
{
    slots = h.numSlots;
    table = h.table;
}

/*=====================================================
~HashTable()      // Destructor
Precondition: None
Postcondition: Deallocates the hash table
=====================================================*/
template <class KeyType>
HashTable<KeyType>::~HashTable()
{
    delete[] table;
    slots = 0;
}

/*=====================================================================
get(const KeyType& k) const   // Return first record with key equal to k
Precondition: Must be a given hash table
Postcondition: Returns a pointer to a record with key k if one exists;
otherwise, return NIL
=====================================================================*/
template <class KeyType>
KeyType *HashTable<KeyType>::get(const KeyType& k) const
{
    int slot = k.hash(slots);
    List<KeyType*> *cur1 = &table[slot];
    Node<KeyType*> *cur = cur1->head;
```

```
    if(cur == NULL)
    {
      throw Empty();
    }

    while(cur != NULL)
    {
      if(*(cur) -> item == k)
        return cur->item;
      else
        cur = cur->next;
    }
    throw Key();
}

/*=============================================================================
insert(KeyType *k)       // Insert record with key equal to k to the hashtable
Precondition: k's value >= 0. Also, if k's value is not already a key in the
hashtable, then the hashtable has space for another record
Postcondition: If the table already had a record with key equal to k's value,
then that record is replaced by k. Otherwise, k has been added as a new record
of the hashtable
=============================================================================*/
template <class KeyType>
void HashTable<KeyType>::insert(KeyType *k)
{
    int slot = k -> hash(slots);
    List<KeyType*> *cur1 = &table[slot];
    Node<KeyType*> *cur = cur1->head;

    cur1->insert(0, k);
}

/*=============================================================================================
remove(const KeyType& k)    // Delete first record with key equal to k from the hashtable
Precondition: Must be a given hash table
Postcondition: If a record was in the hashtable with the specified key k, then that
record has been removed; otherwise the hashtable is unchanged.
=============================================================================================*/
template <class KeyType>
void HashTable<KeyType>::remove(const KeyType& k)
{
    KeyType *temp = get(k);
    int slot = k.hash(slots);
    table[slot].remove(temp);
}

/*=======================================================
toString(int slot) const
=======================================================*/
template <class KeyType>
std::string HashTable<KeyType>::toString(int slot) const
{
    List<KeyType*> *cur1 = &table[slot];
    Node<KeyType*> *cur = cur1->head;

    stringstream s;

    while(cur != NULL)
    {
        s << cur->item->key << ", ";
        cur = cur->next;
    }
```

```
    string returnString = s.str();
    return returnString.substr(0, returnString.size() - 2);
}

/*=======================================================
stream insertion operator overload
=======================================================*/
template <class KeyType>
ostream& operator<<(ostream& stream, const HashTable<KeyType>& ht)
{
  for(int i = 0; i < ht.slots; i++)
  {
    stream << ht.toString(i) << "\n";
  }
  return stream;
}
```

```
// test_hash.cpp
// James Le
// Project 0110
// CS 271 - Data Structures

#include <string>
#include "hash.h"
#include <cassert>

using namespace std;

void test_constructor()
{
  HashTable<Test> test(10);
}

void test_get()
{
  HashTable<Test> test(10);

  Test *str1 = new Test;
  str1->key = "Tokyo";
  test.insert(str1);

  Test *str2 = new Test;
  str2->key = "Seoul";
  test.insert(str2);

  Test find;
  find.key = "Seoul";
  Test *print = test.get(find);
  assert(print -> hash(10) == 5);
}

void test_insert()
{
  HashTable<Test> test(1);

  Test *str1 = new Test;
  str1->key = "London";
  test.insert(str1);

  Test *str2 = new Test;
  str2->key = "Liverpool";
  test.insert(str2);

  Test *str3 = new Test;
  str3->key = "Manchester";
  test.insert(str3);

  assert(test.toString(0) == "Manchester, Liverpool, London");
}

// Test Remove Function
void TestRemove() {
  HashTable<Test> test(1);
  Test *str1 = new Test;
  str1->key = "Amsterdam";
  test.insert(str1);

  Test *str2 = new Test;
  str2->key = "Berlin";
  test.insert(str2);
```

```
  Test *str3 = new Test;
  str3->key = "Paris";
  test.insert(str3);

  assert(test.toString(0) == "Paris, Berlin, Amsterdam");

  Test deleteKey;
  deleteKey.key = "Paris";
  test.remove(deleteKey);
  deleteKey.key = "Amsterdam";
  test.remove(deleteKey);

  assert(test.toString(0) == "Berlin");
}

nt main()
{
  test_constructor();
  test_insert();
  test_get();
  test_remove();

  return 0;
}
```

```
// dict.h
// James Le
// Project 0110
// CS 271 - Data Structures

#ifndef DICTIONARY
#define DICTIONARY

#include <iostream>
#include "hash.h"

template <class KeyType>
class Dictionary : public HashTable<KeyType>
{
public:
  Dictionary(int tableSlots) : HashTable<KeyType> (tableSlots) { } // constructor
  bool empty(); // empty method to check whether the Dictionary is empty or not

  // inhering methods from HashTable class
  using HashTable<KeyType>::get;
  using HashTable<KeyType>::remove;
  using HashTable<KeyType>::insert;
  using HashTable<KeyType>::toString;
  using HashTable<KeyType>::slots;
  using HashTable<KeyType>::table;
};

#endif

#include "dict.cpp"
```

```
// dict.cpp
// James Le
// Project 0110
// CS 271 - Data Structures

#include <iostream>
#include <fstream>
#include <string>
#include <stdib.h>
#include <sstream>

using namespace std;

/*=====================================================================
Empty method
Precondition: A valid Dictionary object
Postcondition: The Dictionary is unchanged, and a boolean is returned.
=====================================================================*/
template <class KeyType>
bool Dictionary<KeyType>::empty()
{
  for(int i = 0; i < slots; i++)
  {
    List<KeyType*> *cur1 = &table[i]; // creating list object
    Node<KeyType*> *cur = cur1 -> head; // creating node within the list object

    if(cur != NULL)
      return false;
  }
  return true;
}
```

```
// test_dict.cpp
// James Le
// Project 0110
// CS 271 - Data Structures

#include "dict.h"
#include <cassert>

using namespace std;

void test_constructor()
{
  Dictionary<Test> test1(5); // constructing Dictionary
}

void test_insert()
{
  Dictionary<Test> test2(1);
  Test *str1 = new Test;
  str1->key = "Madrid";
  test2.insert(str1);

  Test *str2 = new Test;
  str2->key = "Munich";
  test2.insert(str2);

  Test *str3 = new Test;
  str3->key = "Milan";
  test2.insert(str3);

  assert(test2.toString(0) == "Milan, Munich, Madrid");
}

void test_get()
{
  Dictionary<Test> test3(10);
  Test *str1 = new Test;
  str1->key = "Oslo";
  test3.insert(str1);

  Test *str2 = new Test;
  str2->key = "Stockholm";
  test3.insert(str2);

  Test *str3 = new Test;
  str3->key = "Copenhagen";
  test3.insert(str3);

  Test find;
  find.key = "Copenhagen";
  Test *print = test3.get(find); // getting a string from dictionary
  assert(print -> hash(10) == 3) // asserting the get result
}

void test_remove()
{
  Dictionary<Test> test4(1);
  Test *str1 = new Test;
  str1->key = "Budapest";
  test4.insert(str1);

  Test *str2 = new Test;
  str2->key = "Vienna";
```

```
  test4.insert(str2);

  Test *str3 = new Test;
  str3->key = "Prague";
  test4.insert(str3);

  assert(test4.toString(0) == "Prague, Vienna, Budapest");

  Test deleteKey;
  deleteKey.key = "Prague";
  test4.remove(deleteKey);
  deleteKey.key = "Budapest";
  test4.remove(deleteKey);

  assert(test4.toString(0) == "Vienna");
}

void test_empty()
{
  Dictionary<Test> test3(1);

  Test *str1 = new Test;
  str1->key = "Barcelona"; // Inserting a string into a dictionary
  test3.insert(str1);

  assert(test3.toString(0) != "");
}

int main()
{
  test_constructor();
  test_insert();
  test_get();
  test_remove();
  test_empty();

  return 0;
}
```

```
// movie.h
// James Le
// Project 0110
// CS 271 - Data Structures

#ifndef MOVIES
#define MOVIES

#include <iostream>
#include <string>
#include <stdlib.h>
#include <sstream>
#include "dict.h"

class Movie
{
public:
  string title; // string of movie titles
  string cast; // string of cast members

  int hash(int slots) const; // hash function method
  bool operator==(const Movie& mov); // overriding equality operator

  std::string toString() const; // toString method
};

std::ostream& operator<<(std::ostream& stream, const Movie& movie); // ostream operator

#endif
```

```
// movie.cpp
// James Le
// Project 0110
// CS 271 - Data Structures

#include "movie.h"
#include "math.h"

using namespace std;

int Movie::hash(int slots) const
{
  if(title.size() == 1)
  {
    char ch;
    ch = title[0];
    int num = ch;
    return num % slots;
  }

  if(title.size() == 2)
  {
    int asciiCode;
    char cha = title[0];
    char cha1 = title[1];
    asciiCode = cha + cha1;
    int mult = cha;
    return asciiCode * mult % slots;
  }

  if(title.size() > 2)
  {
    unsigned int asciiCode, asciiCode1;
    asciiCode = 1;
    asciiCode1 = 1;
    int n = 77777;
    int numb = 776887;

    for(int i = 0; i < title.size(); i++)
    {
      char c1 = title[i];
      int num1 = c1;
      asciiCode = (asciiCode * num1 * n) % numb; // Prep step for hash function
    }

    for(int j = 0; j < title.size() - 1; j++)
    {
      char c2 = title[j];
      int num2 = c2;
      asciiCode1 = (asciiCode1 * num2 * n) % numb; // Prep step for hash function
    }
    return (asciiCode * asciiCode1) / (title.size() * title.size() * title.size()) % slots; //
 Hash function
  }
}

bool Movie::operator==(const Movie& mov) const
{
  if(title == mov.title)
  {
    return true;
  }
  return false;
```

```
}

string Movie::toString() const
{
  stringstream s;

  s << title << ": ";
  s << cast << '\n';

  string returnString = s.str();
  return returnString.substr(0, returnString.size() - 2);
}

ostream& operator<<(ostream& stream, const Movie& movie)
{
  stream << movie.toString() << "\n";
  return stream;
}
```

```
// query_movies.cpp
// James Le
// Project 0110
// CS 271 - Data Structures

#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
#include <sstream>
#include "movie.cpp"

using namespace std;

int main()
{
  ifstream infile; // file I am reading from
  infile.open("movies_mpaa.txt"); // name of file I am reading from
  ofstream outfile; // file I am outputting in movie frequencies in each slot
  outfile.open("movies_frequencies.txt"); // name of output file

  Dictionary<Movie> movieTable(1000); // create an infinite dictionary that holds movie names
in slots
  string line;

  while(getline(infile, line)) // gets each line
  {
    Movie *movie = new Movie;
    int count = 0;
    while(line[count] != '\t')
    {
      count++;
    }
    movie->title = line.substr(0, count); // create title string from letter 0 up to length of
 title
    movie->cast = line.substr(count + 1); // create cast string from one letter after the titl
e until the end of the line
    movieTable.insert(movie); // inserting the movie object in the dictionary
  }
  infile.close(); // close input file

  for(int i = 0; i < movieTable.slots; i++)
  {
    List<Movie*> *cur1 = &movieTable.table[i]; // create list object
    Node<Movie*> *cur = cur1->head; // create node within the list object

    int length = cur1->length(); // get the length of items in slots, giving us the frequency
of items in each slot
    outfile << length << "\n"; // outputting frequency
  }
  outfile.close(); // close output file

  string movieTitle;
  cout << "Enter a movie title: ";
  getline(cin, movieTitle); // prompting user to enter movie title
  while(true)
  {
    if(movieTitle == "Quit")
    {
      break;
    } else {
      Movie find; // creating a movie object
      find.title = movieTitle;
```

```
      Movie *print = movieTable.get(find); // find user prompted movie title

      cout << endl << "Cast of the movie: " << movieTitle << endl << endl;

      cout << print->cast << endl; // print cast of the movie
      cout << endl;
      cout << "Enter another movie title you want to find or type Quit: ";
      getline(cin, movieTitle); // prompting user to enter another movie title
    }
  }
  return 0;
}
```

```
// test.h
// James Le
// Project 0110
// CS 271 - Data Structures

#ifndef TEST
#define TEST

#include <iostream>

class Test
{
public:
  string key;

  int hash(int slots) const; // Hash function
  std::string toString() const; // toString method
  bool operator==(const Test& tes) const; // Overriding equality operator
};

std::ostream& operator<<(std::ostream& stream, const Test& testt); // ostream operator

#endif
```

```
// test.cpp
// James Le
// Project 0110
// CS 271 - Data Structures

#include "test.h"
#include <string>
#include <stdlib.h>
#include <sstream>
#include <iostream>

using namespace std;

int Test::hash(int slots) const
{
  if(key.size() == 1)
  {
    char ch;
    ch = key[0];
    int num = ch;
    return num % slots;
  }

  if(key.size() == 2)
  {
    int asciiCode;
    char cha = key[0];
    char cha1 = key[1];
    asciiCode = cha + cha1;
    int mult = cha;
    return asciiCode * mult % slots;
  }

  if(key.size() > 2)
  {
    unsigned int asciiCode, asciiCode1;
    asciiCode = 1;
    asciiCode1 = 1;
    int n = 77777;
    int numb = 776887;

    for(int i = 0; i < key.size(); i++)
    {
      char c1 = key[i];
      int num1 = c1;
      asciiCode = (asciiCode * num1 * n) % numb; // Prep step for hash function
    }

    for(int i = 0; i < key.size() - 1; i++)
    {
      char c2 = key[i];
      int num2 = c2;
      asciiCode1 = (asciiCode1 * num2 * n) % numb; // Prep step for hash function
    }
    return (asciiCode * asciiCode1) / (key.size() * key.size() * key.size()) % slots; // Hash
function
  }
}

bool Test::operator==(const Test& tes) const
{
  if(key == tes.key){
    return true;
```

```
  }
  return false;
}

string Test::toString() const
{
  return key;
}

ostream& operator<<(std::ostream& stream, const Test& testt)
{
  stream << testt.toString() << "\n";
  return stream;
}
```